## ⊙ Recursion →

### ⊙ What is recursion ?

- ⊡ When a function calls itself directly or indirectly then it called as recursion.
- ⊡ When a big/complex problems solution depends on a solution of a small problem of same type then we use recursion.

### ⊡ Example →

$2^n$ →

$2^4 = 2 \times 2 \times 2 \times 2$

$2^4 = 2 \times 2^3$

Recurrence Relation ← $2^n = 2 \times f(n-1)$

$f(n)$

### ⊡ Factorial using Recursion →

code →

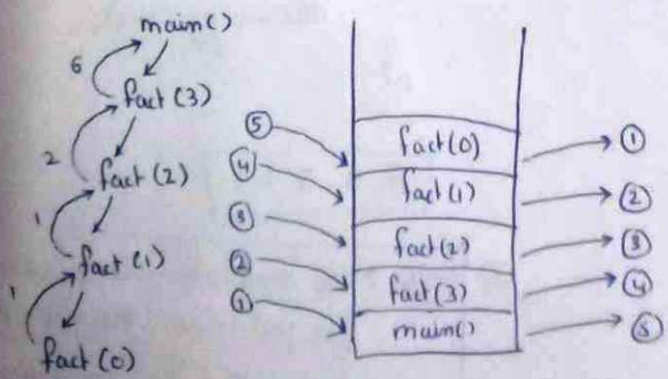
```
int factorial(int n){
    if (n==0){
        return 1;
    }
    return n * factorial(n-1);
}
```

`if (n==0){ return 1; }` → Base Case

`return n * factorial(n-1);` → Recursive Case.

- `n` → bigger problem
- `factorial(n-1)` → smaller problem

### ⊡ Dry Run :→




Ans (6)

### ⊙ Base Case →

- The terminating condition after which our recursive functions stops executing is called as Base case.
- If there is no Base case infinite recursion calls happens in our call stack due to which a time comes when recursion call stack becomes full and starts to overflow, which leads to segmentation fault.

⊙ Head Recursion →

```
Function () {
    Base Case ;
    Recursive Relation ;
    Processing ;
}
```

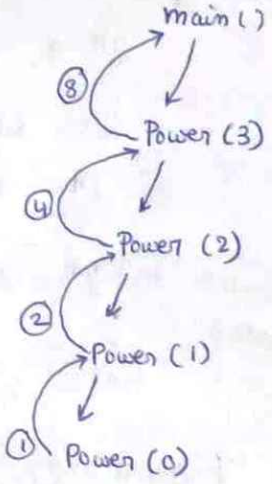
⊙ Tail Recursion →

```
Function () {
    Base Case
    Recursive Relation ;
    Processing ;
}
```

Here processing is before Recursive relation.

⊙ Power of $2^n$ →

```
int Power (int n) {
    if (n==0) {
        return 1;
    }
    int smallerproblem = power (n-1);
    int biggerproblem = 2 * smallproblem;
    return biggerproblem;
}
```




⊙ Print all number in ↑ing order of n →

```
void increasing (int n) {
    if (n==0) {
        return;
    }
    increasing (n-1);
    cout << n << " ";
}
```

1 2 3 4 5.

⊙ Print all number in ↓ing order of n.

```
void decreasing (int n) {
    if (n==0) {
        return;
    }
    cout << n << " ";
    decreasing (n-1);
}
```

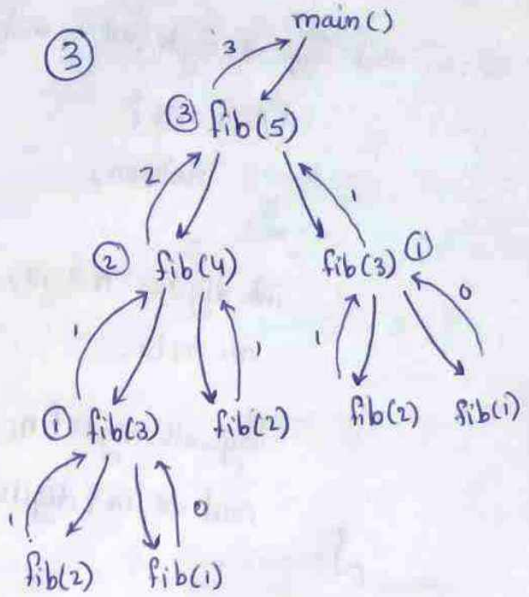
5 4 3 2 1

⊙ Note → We need to handle operation for just 1 case, the rest will be handled by recursion.

① Fibonacci Series

```
int fibonacci(int n){
    if(n==0 || n==1){
        return 0;
    }
    if (n==2){
        return 1;
    }
    return fib(n-1) + fib(n-2);
}
```

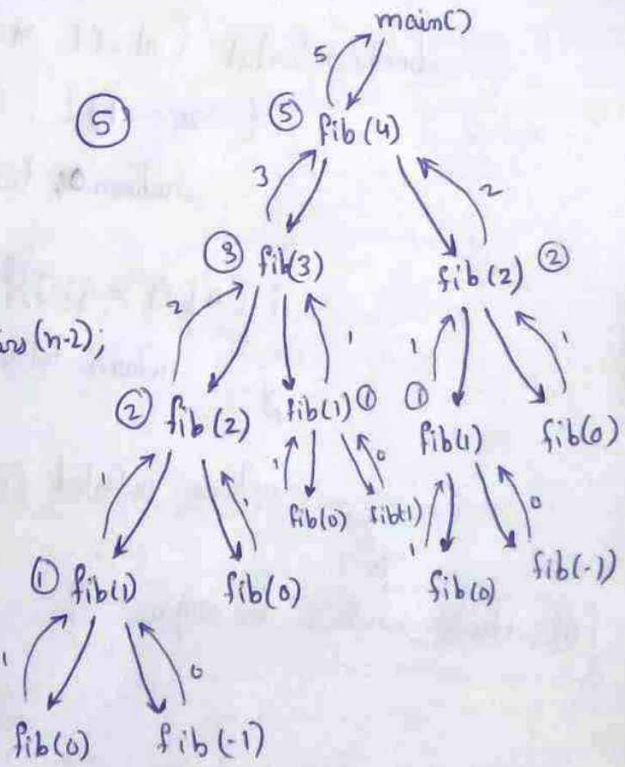
Dry run :




② Count ways to reach Nth Stairs.

```
int climb_stairs(int n){
    if(n<0){
        return 0;
    }
    if (n==0){
        return -1;
    }
    return climb_stairs(n-1) + climb_stairs(n-2);
}
```

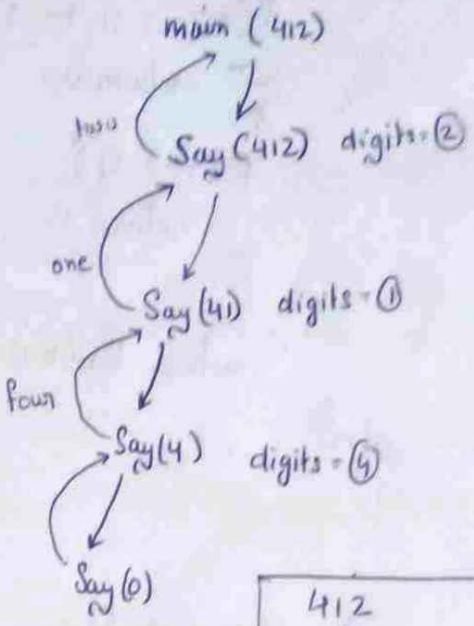
Dry Run :→



## ⊙ Say digits →

```
void Say_all_digits(int n, map<int, string> m){
    if(n==0){
        return;
    }
    int digits = n%10;
    n = n/10;
    Say_all_digits(n, m);
    cout << m[digits] << " ";
}
```

### ⊡ Dry Run →

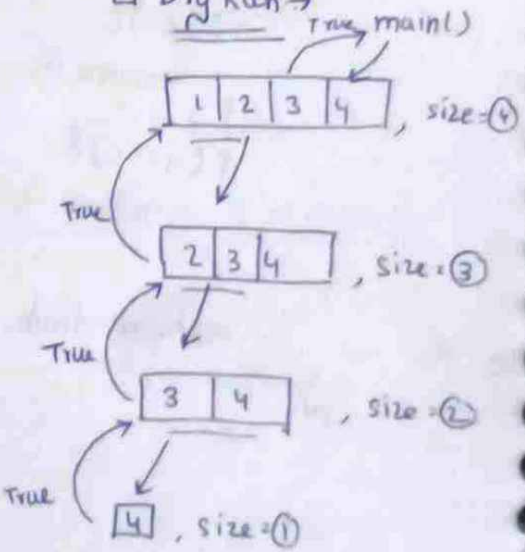



412
⇓
four one two

## ⊙ Check whether an array sorted or not →

```
bool isSorted (int a[], int size){
    if (size==1){
        return true;
    }
    if (a[0] > a[1]){
        return false;
    }
    return isSorted (a+1, size-1);
}
```

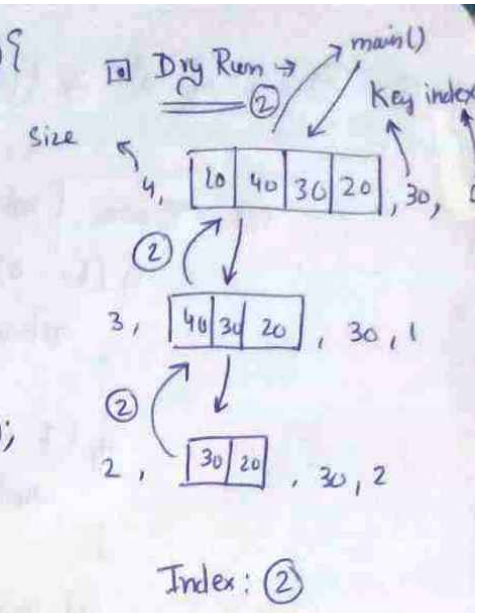
### ⊡ Dry Run →




## ⊙ Check whether an array is

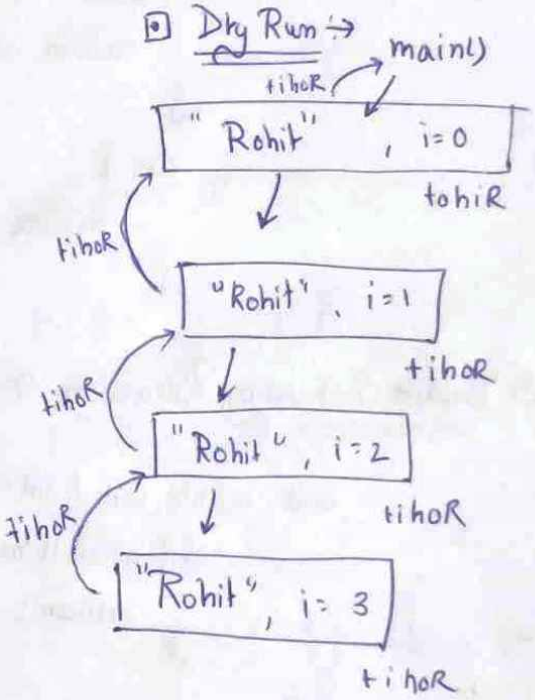
## ⊙ Linear Search using Recursion →

```
int Linear_Search (int a[], int size, int index, int key){
    if( size == 0){
        return -1;
    }
    if ( target == a[0]) {
        return index;
    }
    return Linear_Search ( a+1, size-1, index+1, key);
}
```



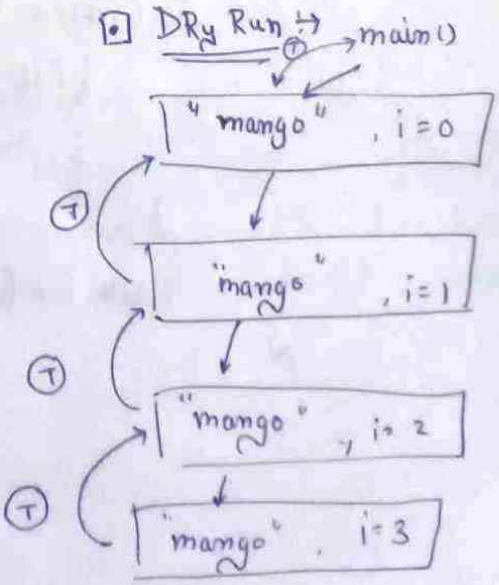

## ⊙ Reverse a String →

```
string Reverse_String (string s, int i) {
    if( i > s.length() - i - 1) {
        return s;
    }
    swap ( s[i] , s[s.length() - i - 1]) ;
    return (s, i+1);
}
```




## ⊙ Pallindrome Check →

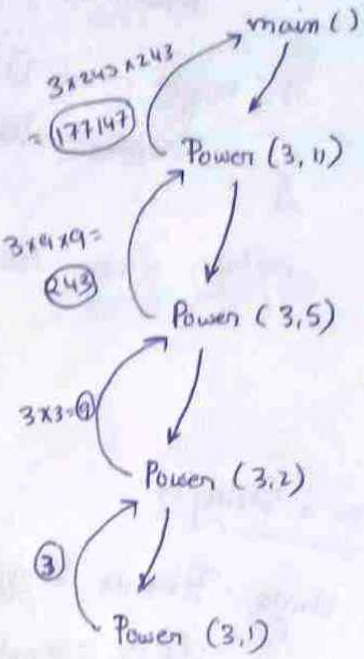
```
bool Pallindrome ( string s , int i) {
    if (i > s.length() - i - 1) {
        return true;
    }
    if ( s[i] != s[s.length() - i - 1]) {
        return false;
    }
    return Pallindrome (s, i+1);
}
```



## ⊙ Power of $a^b$ → (logn solution).

```
int Powers (int a, int b){
    if (b==0){
        return 1;
    }
    if (b==1){
        return a;
    }
    int ans= Powers(a,b/2);
    if(b%2==0){
        return ans * ans;
    }
    else {
        return a * ans * ans;
    }
}
```

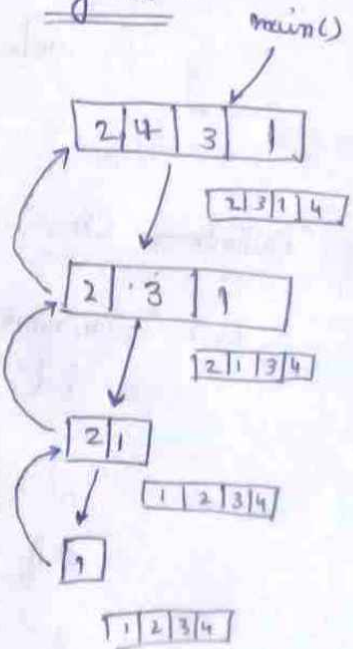
⊡ Dry Run →




## ⊙ Bubble Sort using Recursion →

```
void bubble_Sort (int a[], int n){
    if (n==0 || n==1) {
        return;
    }
    for (int i=0; i<n-1; i++) {
        if (a[i] > a[i+1]){
            swap(a[i], a[i+1]);
        }
    }
    bubble_Sort (a, n-1);
}
```

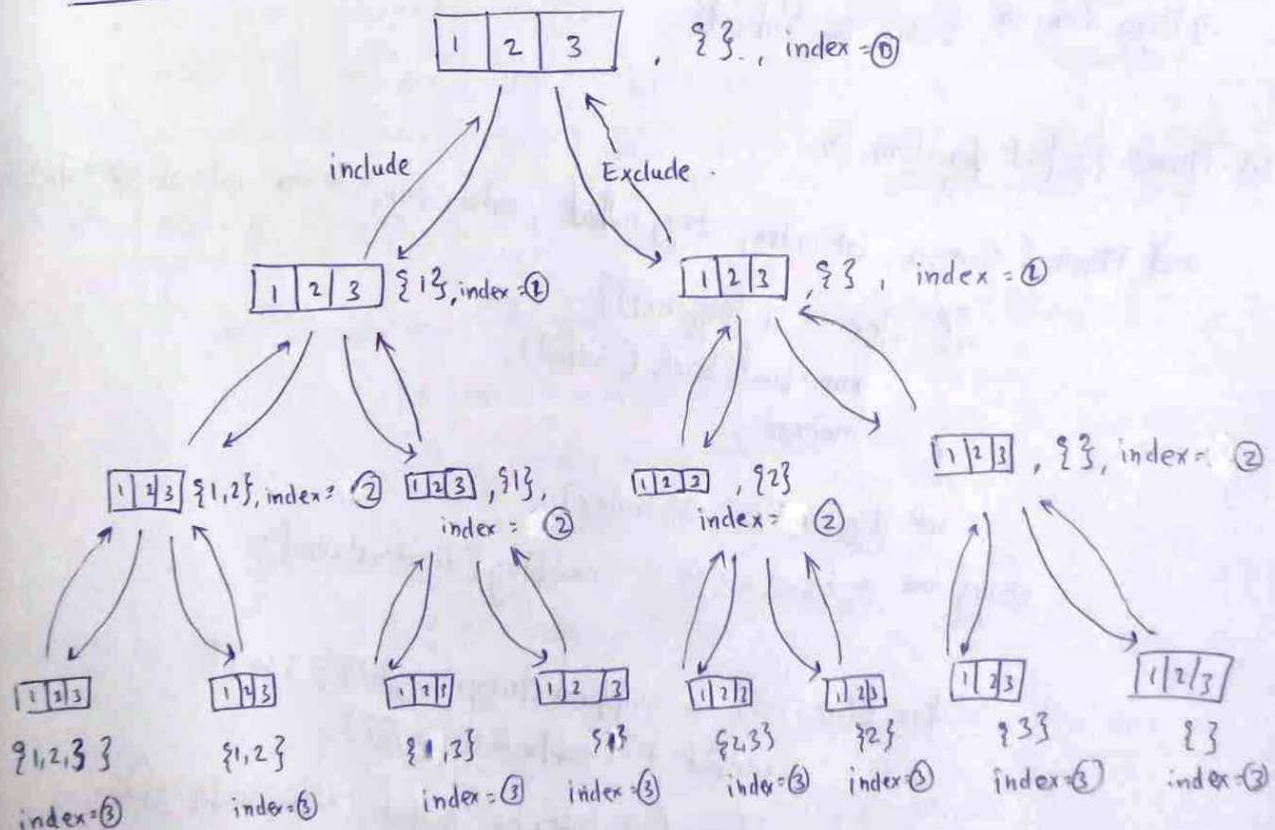
⊡ Dry run →



## ⊙ Subsets →

```
void Subsets ( vector<int> a, int index, vector<int> output, vector<vector<int>> &ans) {
    if (index == a.size()) {
        ans.push_back(output);
        return;
    }
    Subsets(a, index+1, output, ans);   // Exclude
    output.push_back(a[index]);
    Subsets(a, index+1, output, ans);   // Include
}
```

## ⊙ Dry Run



⊙ Subsequences ⇒

```
void Subsets ( string a, int index, string output, vector<string>& ans){
    if( index == a.length() ){
        ans.push_back(output);
        return;
    }
    Subsets ( a, index+1, output, ans); // exclude

    output += a[index];
    Subsets ( a, index+1, output, ans); } // include..
}
```

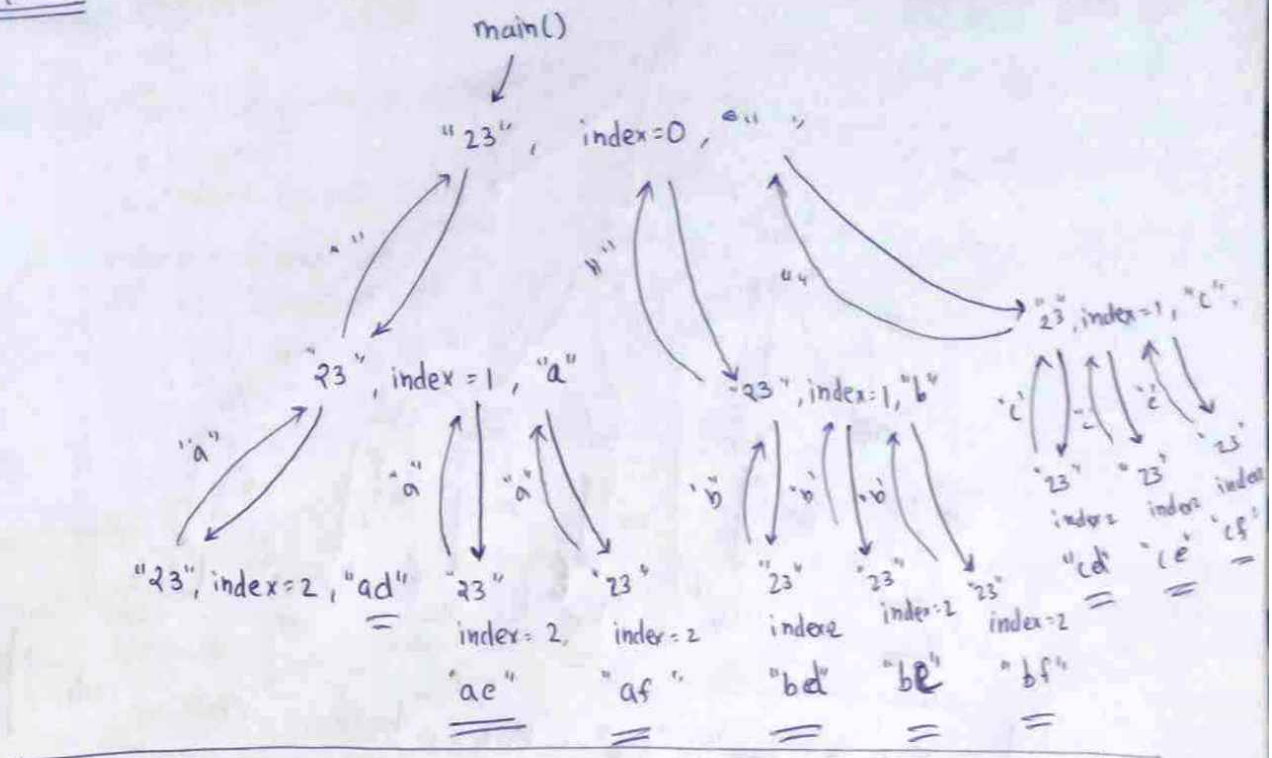
⊙ Dry Run ⇒ Same as Subsets.

⊙ Phone Keypad problem ⇒

```
void Phone ( string a, int index, string output, vector<string>& ans, vector<string> mapping){
    if( index == a.length()){
        ans.push_back (output);
        return;
    }
    int digitvalues = a[index] - '0';
string int mapped_string = mapping[digitvalues];

    for (int i=0; i< mapped string.length(); i++){
        output += mapped string[i];
        Phone (a, index+1, output, ans, mapping);
        output.pop_back();
    }
}
```

## Dry Run →

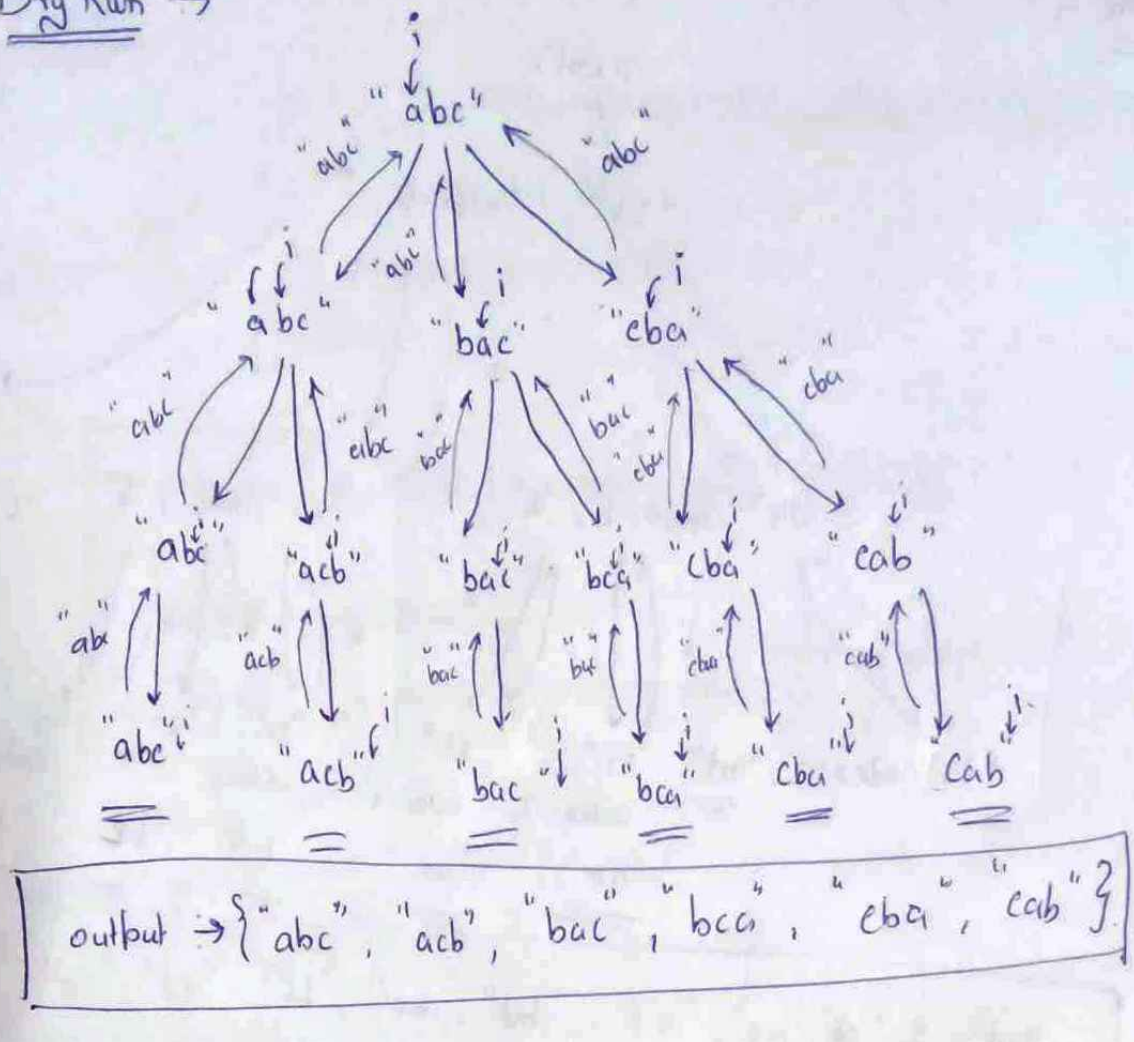



output : { "ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf" }.

## Permutation of Strings →

```
void Permutations (String a, int index, vector <string> & ans) {
    if( index == a.length()) {
        ans.push_back(a);
        return;
    }
    for (int i= index ; i<a.length() ; i++){
        swap (a[i], a[index]);
        Permutations ( a, index+1, ans);
        swap (a[i], a[index]) ; // Backtrack.
    }
}
```

## ⊡ Dry Run →




output → {"abc", "acb", "bac", "bca", "cba", "cab"}

## ⊙ Rat in a maze problem →

```
bool issafe ( int x, int y, vector<vector<int>> m, vector<vector<int>> visited) {
    if ((x>=0 && x<n) && (y>=0 && y<n) && m[x][y]==1 && visited[x][y]==0){
        return true;
    }
    else {
        return false;
    }
}

void solve (vector <vector <int>> & m, int n, vector <vector<int>> & visited, int srcx,
        int srcy, string path, vector <string> & ans) {

    if( srcx = n-1 && srcy = n-1) {
        ans.push_back (path);
        return;
    }
```

```
// Down
int newx = srcx+1;
int newy = srcy;
if(issafe(newx, newy, n, m, visited)){
        path += 'D';
        solve(m, n, visited, newx, newy, path, ans);
        path.pop_back();
}
```

// Left

```
newx = srcx;
newy = srcy-1;
if(issafe(newx, newy, n, m, visited)){
        path += 'L';
        solve(m, n, visited, newx, newy, path, ans);
        path.pop_back();
}
```

// Right

```
newx = srcx;
newy = srcy+1;
if(issafe(newx, newy, n, m, visited)){
        path += 'R';
        solve(m, n, visited, newx, newy, path, ans);
        path.pop_back();
}
```

// Up

```
newx = srcx-1;
newy = srcy;
if(issafe(newx, newy, n, m, visited)){
        path += 'U';
        solve(m, n, visited, newx, newy, path, ans);
        path.pop_back();
}
```

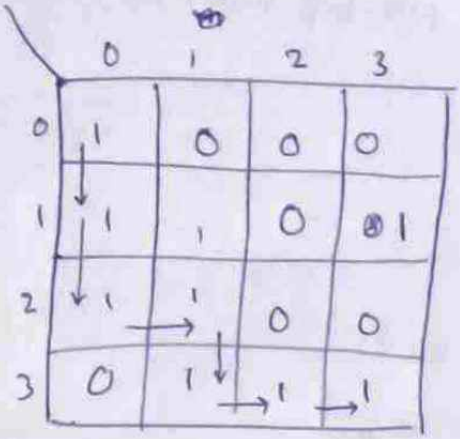
visited [srcx] [srcy] = 0;

}

## ⊡ DryRun. :→

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | ✓1 | 0 | 0 | 0 |
| 1 | ✓1 | 0 | 0 | 0 |
| 2 | ✓1 | ✓1 | 0 | 0 |
| 3 | 0 | ✓1 | ✓1 | ✓1 |

visited.



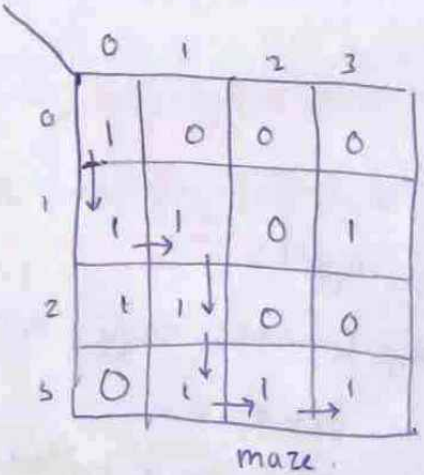
maze

L: Left
D: Down
R: Right
U: Up.

path: DDRDRR

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | ✓1 | 0 | 0 | 0 |
| 1 | ✓1 | ✓1 | 0 | 0 |
| 2 | 0 | ✓1 | 0 | 0 |
| 3 | 0 | ✓1 | ✓1 | ✓1 |

visited



maze.

path: DRDDRR

ans: { "DDRDRR", "DRDDRR" }.

## ⊙ Backtracking →

- ⊡ In Backtracking, we are checking all possible paths and then see whether we get solution or not.
- ⊡ Moreover, the path which is visited once which has got no solution is discarded. We won't visit that path again.

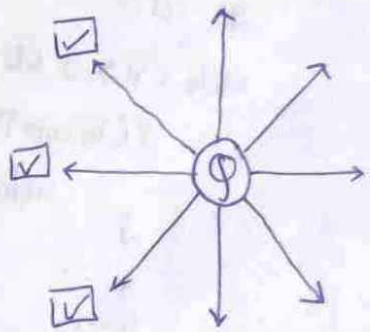
## ⊙ Rate in a maze Problem →

// Same as in Recursion.

## ⊙ N-Queen Problems →

⊡ Conditions →

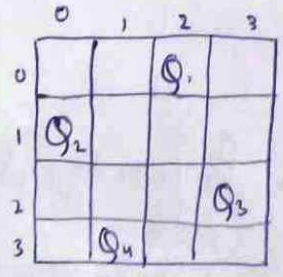
(i) Each row must have 1 Queen.
(ii) Each column must have 1 Queen.
(iii) No two Queens can attack each other.



Q: Queen → 8 direction Attack.

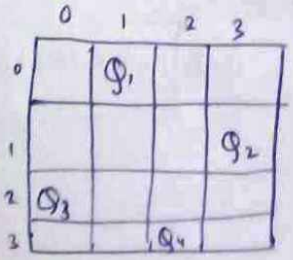
☑: This movement of Queen is needed to be checked.

⊡




Here, not any queen can attack the other queen.

if input is 4 we have got 2 possible ways.



⊙ Code →

```
bool is_safe (int row, int col, vector<vector<int>> boards, int n) {
    int x = row;
    int y = col;
    while (y >= 0) {
        if (boards[x][y] == 1) {
            return false;
        }
        y--;
    }
```

→ for Row.
↓
can be optimised using map
check map < col → T/F >

```
    x = row;
    y = col;
    while (x >= 0 && y >= 0) {
        if (boards[x][y] == 1) {
            return false;
        }
        x--;
        y--;
    }
```

→ for upper diagonal.
↓
can be optimised using map.
check map < col+row → T/F >

```
    x = row;
    y = col;
    while (x < n && y >= 0) {
        if (boards[x][y] == 1) {
            return false;
        }
        x++;
        y--;
    }
    return true;
}
```

→ for lower diagonal.
↓
can be optimised using map
check map < (n-1 + col-row) → T/F >

```
void solve (int col, int n, vector<vector<int>>& boards, vector<vector<int>>& ans){
    if (col == n){
        store (n, boards, ans);
        return;
    }
    for (int row = 0; row < n; row++) {
        if (is_safe (row, col, boards, n)) {
            boards [row] [col] = 1;
            solve (col+1, n, boards, ans);
            boards [row] [col] = 0;
        }
    }
}
```

- Time Complexity → O(N!)
- Space Complexity → O(N * N).

⊙ Sudoku Solver →

→ ⊡ Important conditions :

(i) → 1 row → 1-9 digits ⇒ Exactly once appear

(ii) → 1 column → 1-9 digits ⇒ Exactly once appear

(iii) → 3x3 grid → 1-9 digits ⇒ Exactly once appear

⊙

- Time Complexity ⇒ $9^m$
- Space Complexity ⇒ O(1)

## Code →

```
bool is_safe (int val, int row, int col, int n, vector<vector<int>>& board){
    for(int i=0; i<n; i++){
        if( board [row] [i] == val){
            return false;
        }
        if ( board [i] [col] == val){
            return false;
        }
        if ( board [3*(row/3) + i/3] [3*(col/3) + i%3] == val){
            return false;
        }
    }
    return true;
}

bool solve ( vector<vector<int>>& board, int n){
    for(int row=0; row<n; row++){
        for(int col=0; col<n; col++){
            if(board [row] [col] == 0){
                for( int val=1; val<=9; val++){
                    if( is_safe ( val, row, col, n, board)){
                        board [row][col] = val;
                        bool futher possibility = solve (board, n);
                        if (futher possibility){
                            return true;
                        }
                        else{ board [row] [col] = 0;
                        }
                    }
                }
                return false;
            }
        }
    }
    return true;
}
```

checking column ←

checking row ←

checking grid ←